Logic–based Genetic Programming with Definite Clause Translation Grammars

Brian J. ROSS

Brock University, Dept. of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
`bross@cosc.brocku.ca`

Received 22 June 1999
Revised manuscript received 20 April 2000

Abstract DCTG-GP is a genetic programming system that uses definite clause translation grammars. A DCTG is a logical version of an attribute grammar that supports the definition of context–free languages, and it allows semantic information associated with a language to be easily accomodated by the grammar. This is useful in genetic programming for defining the interpreter of a target language, or incorporating both syntactic and semantic problem–specific contraints into the evolutionary search. The DCTG-GP system improves on other grammar–based GP systems by permitting non–trivial semantic aspects of the language to be defined with the grammar. It also automatically analyzes grammar rules in order to determine their minimal depth and termination characteristics, which are required when generating random program trees of varied shapes and sizes. An application using DCTG-GP is described.[*1]

Keywords Genetic Programming, Evolutionary Computation, Definite Clause Translation Grammar, Logic Programming, Stochastic Language Inference.

§1 INTRODUCTION

Genetic programming (GP) implementations have benefitted from simple program denotations. In Koza's denotation of program trees as implemented

[*1] *New Generation Computing*, vol.19, n.4, 2001, pp. 313-337.

by S-expressions in Lisp, a simple context–free language with one nonterminal is used [24]. This basic grammatical definition of program structure makes program generation and reproduction straight-forward to implement and efficient to execute by the GP system. The only grammatical requirement is that all expressions are either single terminals, or a single nonterminal followed by zero or more arguments – each of which is defined recursively. Another potential advantage of this scheme is that it permits a rich search space to be used during evolution, since a wide syntactic variety of trees is permissible. The user must ensure the closure of all such trees, so that every program can be executed without errors.

Although robust, this simple treatment of program structure is sometimes detrimental. Some GP systems require distinguished branches in the program tree representation, and these special cases complicate implementations. For example, ADF definitions require specially denoted ADF branches [25]. More importantly, more rigorous parse tree representations using data types and grammars is beneficial to evolution efficiency [31, 41, 43]. One reason for this is that the basic S-expression representation does not necessarily favour semantically useful programs during evolution. This effect becomes more critical for target languages that have more restricted syntax than Lisp. In addition, the use of grammars permits problem domain specific syntactic constraints to be injected in the evolved program structure, thus enhancing evolution effectiveness by refining the search space.

DCTG-GP is a logic–based genetic programming system that uses a definite clause translation grammar (DCTG). A DCTG is a logic grammar which permits Backus–Naur style specification of context–free grammars. DCTG's also permit multiple semantic definitions to be included within the definition of the grammar, allowing the syntactic definition of the language to be unified with various semantic properties. This is beneficial for GP systems in a number of ways. For example, the complete computational semantics of a language may be defined with the DCTG in order to define a complete run–time environment for the language, instead of a separate interpreter or compiler. Alternatively, the DCTG semantic rules can define semantic properties which simplify the target language's grammar. Given that problem–oriented syntactic constraints can aid evolution, the use of the DCTG semantics for refining these constraints can be very convenient. Finally, depending on the problem, it is possible to partially interpret programs within the DCTG in order to determine their admissibility, and hence within tree generation and reproduction processes during evolution.

GP systems must generate random trees during initial population creation and program mutation. The sizes of trees must be controlled during randomized generation, in order to ensure that they have a variety of different shapes in the initial population. It is difficult, however, to control tree sizes as generated by general context–free grammars. To overcome this problem, DCTG-GP uses a grammar pre–processor, which analyzes DCTG production rules in order to obtain their recursion characteristics and the minimal depths of subtrees generated by them. This information is required for the effective generation of program trees of desired depths during the initial population genesis and program mutation.

Section 2 reviews genetic programming and definite clause translation grammars. The DCTG-GP system is presented in Section 3. An example application using DCTG-GP is described in Section 4. DCTG-GP is compared with other grammatical–based GP systems in Section 5. Section 6 concludes the paper with a discussion.

§2 BACKGROUND

2.1 Genetic Programming

Genetic programming [4, 24)] is a method of automatic programming in which programs are evolved using a genetic algorithm [18)]. Both GP and GA are characterized by their use of the following (see Fig. 1): (i) an initial population of random individuals, which in the case of GP are randomly–constructed programs; (ii) a finite number of generations, each of which results in a new or replenished population of individuals; (iii) a problem–dependent fitness function, which takes an individual and gives it a numeric score indicative of that individual's ability to solve a problem at hand; (iv) a fitness–proportional selection scheme, in which programs are selected for reproduction in proportion to their fitness; (v) reproduction operations, usually the crossover and mutation operations, which take selected programs and generate offspring for the next generation.

The essential difference between GP and GA is the denotation of individuals in the population. A pure GA uses genotypes that are fixed–length bit strings, and which must be decoded into a phenotype for the problem being solved. A GP uses a variable–length computer program genotype, which is directly executable by some interpreter (typically Lisp). The use of program-

```
1. Initialize: Generate initial randomized population.
2. Evolution:
   GenCount := 0
   Loop while GenCount < maximum generations
      and fitness of best individual not considered a solution {
         Loop until New population size = max. population size {
            Select a genetic operation probabilistically:
            → Crossover:
                  Select two individuals based on fitness.
                  Perform crossover.
            → Mutation:
                  Select one individual based on fitness.
                  Perform mutation.
            Add offspring to new population.
         }
      GenCount := GenCount+1
   }
3. Output: Print best solution obtained.
```

Fig. 1 Genetic Algorithm

ming code as genotype is powerful, as its generality permits a vast number of problems to be addressed. It also means that GP is the closest realization yet for true automatic programming, in which computers program themselves after being given a description of a problem in which to solve.

The two main reproduction operators used in GP are crossover and mutation. Crossover is the more important of the two operators, as it permits the genetic combination of program code from programs into their offspring, and hence acts as the means for inheritance of desirable traits during evolution. Crossover takes two selected programs, finds a random *crossover point* in each program's internal representation (normally a parse tree), and swaps the subtrees at those crossover points. The crossover points must be selected so that the resulting offspring are syntactically correct. With Koza's simple program structure [24)], if a subtree with a nonterminal root (function label) is selected in one program, it should be swapped by a subtree with a nonterminal root from the other program. Mutation is not as commonly performed in GP. When it is used, a random mutation point is found in a selected program, and the subtree at that node is replaced with a new, randomly generated tree. This is the means by which new genetic traits can be introduced into the population during evolution. Although crossover and mutation preserve the grammatical integrity of programs, the user must ensure *closure* – that the resulting programs are always executable. So long as closure is maintained, all programs derivable by the GP system will be executable by the fitness function, and hence their fitness will be derivable.

2.2 Definite Clause Translation Grammars

DCTG's were invented as a means for specifying the syntax and semantics of languages in a convenient, unified fashion [1]. A DCTG has a syntactic component, and possibly multiple semantics components. The syntactic component is equivalent to that used by definite clause grammars [32], and permits the definition of context–free languages using a Backus–Naur style of syntax. Since DCTG's and DCG's are logic grammars that are usually implemented in Prolog [8], the syntactic component permits user–defined arguments, embedded Prolog goals, and calls to DCTG semantic rules to be used within nonterminal productions. These semantic references represent context–sensitive information about the language, and can verify grammatical constructs for semantic viability during generation or parsing.

Some definitions are now given. (See [19] for more information on the fundamentals of formal grammars.) A context–free grammar (CFG) is a 4–tuple (N, Σ, P, S), where N are nonterminals, Σ are terminals, P are productions of the form $x \rightarrow y$ $(x \in N, y \in (N \cup \Sigma)^*)$, and S is a start symbol. A *derivation step*,

$$\alpha A \beta \overset{P_i}{\rightarrow} \alpha \gamma \beta$$

represents the application of production $P_i : A \rightarrow \gamma$ to nonterminal A. Derivations are conveniently denoted by *derivation trees* or *parse trees*, in which the root is S, the internal nodes are N, leafs are Σ, and the descendents of N nodes represent the application of particular productions (ie. each nonterminal node is associated with a corresponding production). Note that a nonterminal can have multiple productions applicable to it.

Using logic programming terminology, a *goal* is a nonterminal or terminal reference in the right–hand side of a production. The set of goals in a production is the *body*. Each production has a *head*, which includes a reference to the nonterminal the production encodes, as well as optional data arguments. A *(logic) variable* denotes a place holder for an expression, which is normally a data structure element when it appears within a head or goal.

The syntactic form of DCTG rules is:

$$\begin{array}{l} H \ ::= \ B_1, \ B_2, \ ..., \ B_k \\ <:> \\ S \ ::- \ G_1, \ G_2, \ ..., \ G_n. \end{array}$$

The line with head H defines the syntactic definition for the production, for some particular nonterminal encoded in H. The B's are goals of the production. They may refer to nonterminals, or terminals (encoded as Prolog list elements). They may also be calls to Prolog code, or references to semantic components elsewhere in the grammar; such goals are embedded within braces. The symbol "<:>" delimits the syntactic rule from the semantic definition with head S. There may be more than one semantic definition per production. The goals G are references to either semantic goals in the grammar or Prolog goals. When such a goal is a semantic reference, it specifically refers to a semantic goal from one of the syntactic goals B from the syntactic production above it. The "^^" operator, used in concert with a variable, links a semantic goal with a syntactic goal:

$$\begin{array}{l} H \ ::= \ ..., \ B\text{^^}N, \ .... \\ <:> \\ S \ ::- \ ..., \ N\text{^^}G, \ .... \end{array}$$

Here, the goal G in the semantic definition refers to the semantic component associated with the parse tree for nonterminal B in the syntactic definition. (Terminals do not have productions associated with them, and cannot have semantic rules). This scheme permits access to various diverse subtrees of the overall parse tree, and the semantic components associated with those subtrees.

A typical DCTG production for natural language processing is the following:

```
verb_phrase ::= verb^^V, { transitive(V) }, noun_phrase^^N
<:>
(agree(Num) ::- V^^agree(Num)),
(logical_form(X,P) ::- V^^logical_form(transitive,X,Y,P1),
                       N^^logical_form(Y,P1,P)).
```

The rule `verb_phrase` is the syntactic definition, and defines a single CFG production stating that a verb phrase is a verb followed by a noun phrase. The intervening reference to `transitive` is an embedded Prolog goal, and is used to verify that the verb referred to by `V` is transitive. This is context–sensitive information, since it is used only when the rule is interpreted with a user's sentence, and it must evaluate to true if the sentence is to be considered grammatically correct. There are two semantic rules associated with this production. The rule `agree` generates the agreement class for the verb phrase, based upon the agreement class of the verb used within it, for example, a singular or plural verb. The `logical_form` rule generates a logical representation for this portion of the parse tree, the details of which are done recursively by the calls to `logical_form` within the verb and noun phrase portions of the parse tree.

§3 SYSTEM DESIGN

DCTG-GP is genetic programming system implemented in SICStus Prolog 3.8.1 [39] on Silicon graphics IRIX 6.3 and Windows 98 platforms. The DCTG library is authored by Harvey Abramson [1]. The DCTG-GP engine uses conventional GP strategies, such as steady–state or generational evolution, tournament selection, and local search. Parameters for GP control are outlined in Section 4. The remainder of this section will discuss features particular to the grammatical and logical foundation of DCTG-GP.

3.1 Grammar Representation

The syntactic and semantic definitions of the DCTG grammar are initially parsed by DCTG library utilities and translated to standard Prolog clauses. When executed, these clauses parse argument lists of tokens according to the rules of the grammar. The clauses also permit generation of sentences conforming to the grammar. An internal tree structure is used by DCTG-GP to concisely represent the parse trees of programs. Nonterminal nodes have the form,

$$node(Call,\ [Node_1, \cdots, Node_k],\ ID)$$

while leaf nodes are represented as,

$$[constant].$$

The nonterminal structure represents a single internal node in the parse tree. *Call* is a copy of the head of the corresponding grammar production for the rule, and the exact DCTG rule used is identified by an ID number. The list of nodes are immediate descendents of this node as encoded in the production body, and each $Node_i$ recursively uses this same node representation. This representation does not include embedded references to semantic goals and Prolog code goals. If such embedded goals are used, the DCTG rules are interpreted in parallel with the interpretation of the parse tree representation. We call this parallel interpretation *verification*. Should embedded goals fail during verification, then the parse tree will be invalid, and the tree generation or reproduction step will fail. The GP engine will then reattempt the evolutionary action being undertaken.

3.2 Grammar Preprocessor

The DCTG's used by DCTG-GP are generative grammars, and the random selection of DCTG rules during derivation will generate grammatically–correct random programs of various shapes and sizes. Any embedded semantic goals and Prolog goals in the rule bodies are executed during this random generation, which acts as additional context–dependent verification of the parse tree being constructed. Tree generation happens during the initial population generation, as well as during any mutation steps during evolution.

GP runs commence by generating initial populations of random programs. The conventional wisdom is that evolution is most effective when the initial population of programs has a variety of shapes and sizes. One popular means for deriving such populations is Koza's ramped half–and–half generation strategy [24]. Half the programs generated are *full trees*, in which trees are generated through a range of maximal depths, and each branch of the tree has a maximal depth. The other half of the population are *grow trees*, in which a range of tree depths are also generated, but where each tree's branch must be no greater than the maximal depth. Generating a tree of a desired depth is easy with Koza's S-expression grammar. The grammar specifies that expressions are either terminals, or a nonterminal followed by expression arguments. Therefore, derivations can be terminated at any time, merely by using terminals instead of nonterminals.

Unfortunately, deriving trees having desired depths is not trivial to do with a grammar. When a tree is being derived, each nonterminal node is to be expanded via one of many productions for that nonterminal. It is difficult to ensure that a tree will have a desired depth because it cannot be directly determined whether the application of a particular production will result in a branch of required depth. For example, some productions may terminate immediately should their bodies consist of terminals. Other productions may recurse, which in itself may cause infinite recursion. This problem with generative grammars is well known, and has been investigated elsewhere [2, 5, 21].

The solution given here, although sharing characteristics with other algorithms, uses a simple algorithm which yields acceptable results, albeit not as precisely as other more complex approaches. To generate a ramped half–and–half population with the DCTG, some analyses of the grammar is first undertaken, in order to determine two characteristics of rules: (i) the minimal depth of trees generated by each DCTG rule; and (ii) whether rules may generate terminating (finite) or arbitrarily deep (recursive) trees. Once these characteristics are found, rules can be selected more prudently. The first characteristic identifies rules that can terminate branches at appropriate depths, and the second characteristic identifies rules that will more likely result in trees with fuller branches.

The following notation is used in the algorithm pseudocode. Let R_i $(1 \leq i \leq K)$ denote the K nonterminal labels in the grammar. Let $R_{i,j}$ $(1 \leq i \leq K,\ 1 \leq j \leq N_i)$ denote the individual production rules in the grammar, where i indexes the K nonterminals, and each nonterminal has N_i production rules defining it. The goals in each $R_{i,j}$ are indexed as $G_{i,j,k}$. For example,

$$\begin{array}{lcl}
p :: -a, b, c. & & R_{1,1} :: -G_{1,1,1},\ G_{1,1,2},\ G_{1,1,3}. \\
p :: -d, a. & & R_{1,2} :: -G_{1,2,1},\ G_{1,2,2}. \\
p :: -e. & \Rightarrow & R_{1,3} :: -G_{1,3,1}. \\
q :: -e, f. & & R_{2,1} :: -G_{2,1,1},\ G_{2,1,2}. \\
q :: -f. & & R_{2,2} :: -G_{2,2,1}.
\end{array}$$

In the above, the nonterminal p is denoted R_1, and is defined by the rules $R_{1,1}$, $R_{1,2}$, and $R_{1,3}$. The goals g above may refer to nonterminals (R_i's) or terminals.

The first analysis determines the minimal depth of subtrees generated by each grammar rule $R_{i,j}$. There may be multiple rules applicable for a particular nonterminal R_i, and these rules may generate trees of varying minimal depths. When generating trees of a given depth, selected rules must be assured of generating trees within some depth limit. The algorithm for computing minimal rule depth is in Fig. 2. It is an iterative algorithm which processes the set of grammar rules in a bottom-up fashion. Two values are determined in the algorithm: (i) $RuleDepth_{i,j}$ is the minimum tree depth for each rule $R_{i,j}$; and (ii) $NonTermDepth_i$ is the minimum tree depth for each nonterminal R_i. Note that $RuleDepth_{i,j}$ is used by GP later, while $NonTermDepth_i$ is used only

```
iterate over $R_{i,j}$ until $RuleDepth_{i,j}$ found for all rules
   or no further progress possible {
      Examine a non-processed rule $R_{i,j}$:
         1a. if all $G_{i,j,k}$ are terminals
               then $RuleDepth_{i,j} = 1$
         1b. else if each nonterminal $G_{i,j,k}$ has a known $NonTermDepth_n$
               then $RuleDepth_{i,j} = 1 + maximum(NonTermDepth_n)$
         2. if a new nonterminal $R_i$ had $RuleDepth_{i,j}$ calculated in step 1
              then $NonTermDepth_i = RuleDepth_{i,j}$
}
```

Fig. 2 Minimal Depth Algorithm

during the run of the minimal depth algorithm. The base case is a rule containing only leaf nodes, which has a minimal depth of 1 (step 1a). The iterative case is a rule whose nonterminals all have known minimal depths; the minimal depth for this rule is then one level deeper than the maximum depth of all the nonterminal subtree depths (step 1b). Additionally, the first time a minimal depth $RuleDepth_{i,j}$ is determined for any rule comprising a nonterminal R_i, it becomes the minimal depth $NonTermDepth_i$ for R_i as referred to by all other rules (step 2). Processing continues until either all rules have minimum depths computed for them, or rules with undeterminable minimal depth are discovered. These rules are recursive ones that have no means of termination, and must be fixed by the user before proceeding. For example, if the rule

$$r :: -r.$$

is used, and there is no other production for r, then r is obviously infinitely recursive, and is rejected by the system.

An example run of the minimal depth algorithm is as follows. Consider the grammar in Fig. 4. The goals enclosed in brackets "[ ]" and braces "{ }" are treated as terminals. In the following, the notation $expr_3$ denotes of the third rule of *expr*.

```
iterate over $R_{i,j}$ until all $R_{i,j}$ labelled
   or no further progress possible {
      Examine a non-labelled rule $R_{i,j}$:
         1. if all $G_{i,j,k}$ are terminals
            or all $G_{i,j,k}$ nonterminals are labelled terminating
            then $R_{i,j}$ labelled terminating.
         2. if all $R_{i,j}$ for an $R_i$ labelled as terminating
            then label $R_i$ as terminating.
}
Label remaining unlabelled rules nonterminating.
```

Fig. 3 Rule Termination Algorithm

Iteration	RuleDepth	NonTermDepth
1	*guardedexpr*$_1$=1, *guardedexpr*$_2$=1, *intval*$_1$=1, *probval*$_1$=1	*guardedexpr*=1, *intval*=1, *probval*=1
2	*expr*$_1$=2, *choice*$_1$=2	*expr*=2, *choice*=2
3	*expr*$_2$=3, *expr*$_3$=3, *expr*$_4$=3, *choice*$_2$=3	

The second analysis determines termination (recursion) characteristics of rules. A rule is said to be *terminating* if neither it nor its descendents can result in recursive nonterminal derivations, and hence always derives finite trees. Otherwise, the rule is *nonterminating.* This knowledge about the recursive properties of rules is needed during the generation of full trees, which requires that branches be as deep as possible. In order to generate full trees, the random selection of DCTG rules should be from those that enable the deepest branches possible. It can be counterproductive to select a rule that always generates a finite-depth subtree, since the tree may be forced to terminate at a shallower depth than desired. Knowing which rules have recursive descendents is therefore valuable, as these rules are more likely to generate trees having a required maximal depth.

The algorithm for determining the termination properties of rules is in Fig. 3. This algorithm is also an iterative, bottom–up one. The algorithm determines whether each rule is to be labelled *terminating* or *nonterminating.* The user may override the termination designations for a nonterminal, because it is not useful to generate deep program structures for some language constructs. For example, generating a deep enumerated Pascal set structure may mean that the generated set will always include all the members of that enumerated space, which is probably undesirable.

Consider once again the grammar in Fig. 4. During the first iteration, $guardedexpr_1$, *intval* and *probval* are determined to be *terminating.* However, in the next iteration, all the remaining rules refer to goals which are not yet classifiable as terminating, and hence no changes occur. The final step in the algorithm will therefore label the rest of the rules as *nonterminating*, which indicates that they can generate recursive, potentially nonterminating trees.

The above method for determining minimal rule depths and recursion in rules is based solely on nonterminal label analysis within productions. The algorithms give an approximation of the minimal size tree derivable with each production, as well as the recursive nature of each production. In reality, the actual minimal depths of productions is complex to determine, because not every production for a nonterminal is applicable during the derivation, due to the existence of arguments values, embedded semantic and Prolog goals used within the DCTG rules. Although more in-depth consideration of these features could be undertaken, (for example, using abstract interpretation 9)), we ignore them for the sake of efficiency and practicality. This design decision is supported by the fact that argument values are usually dependent upon run–time data that is not available during population generation. The main goal is that trees generated by the DCTG are more likely to have desired shapes and sizes, rather than guarantee that they precisely will.

3.3 Reproduction Operations

A requirement for crossover and mutation operations is that offspring are grammatically correct with respect to the DCTG. As is suggested elsewhere 16, 41), maintaining grammatical integrity during crossover is ensured by selecting subtrees from each parent that have roots of the same nonterminal type. Terminals are never selected for crossover, because they are unlabelled within the DCTG parse tree, and hence cannot be easily identified for correctness. (If the user wishes the selection of terminal information during crossover, then nonterminal rules should be defined to represent them.) Similarly, only subtrees with nonterminal roots can be selected for mutation. A tree with the same nonterminal type as the selected subtree is generated, and it replaces the selected subtree.

If the DCTG productions contain semantic goals or Prolog goals, they will need to be interpreted after crossover and mutation in order to verify the correctness of the parse tree. This is considered a verification step performed upon the offspring. In applications that only use the DCTG's syntactic definitions, and do not use semantic rules nor embedded Prolog goals, semantical verification is unnecessary and is omitted.

Crossover and mutation can fail in a number of ways. During crossover, the nonterminal type of a selected subtree in one parent may not exist in the other parent. The offspring may exceed maximum depth bounds. Semantic verification may fail. If any of these situations occur, then the reproduction step is retried on the given parents up to a user-specified number of times. If all reattempts fail, then that reproduction operation fails for the selected parent(s), and evolution continues with the selection of other genotypes for reproduction.

§4 AN EXAMPLE EXPERIMENT

This section presents an example application of DCTG-GP. The experiment involves the grammatical inference of a probabilistic pattern matching language. The experiment is in the spirit of work elsewhere [35], which should be consulted for additional details and examples. The given experiment is intended to show two specific practical benefits of DCTG-GP: (i) the encoding of syntactic contraints, and how such constraints are unwieldy to implement in non-DCTG systems; and (ii) the encoding of operational semantics. It is not the intent of this section to study the effects of DCTG-GP on evolutionary performance, nor compare the inference undertaken with other inference paradigms, which may or may not be more effective for this particular inference problem. Therefore, the experimental details and results in Sections 4.4 and 4.5 are not of primary importance here, but are included for completeness. Furthermore, this section does not explore the variety of languages that can be encoded with a DCTG; please refer to Section 5 for an overview of work in grammatical GP and related topics.

An overview of the problem is first given. The implementation of the probabilistic language in DCTG-GP is then discussed, which illustrates some major strengths of the system. Experimental parameters and strategies are outlined, and the overall results of the experiment are given. The reader should have some exposure to formal language theory [19].

4.1 Probabilistic Pattern Matching

Grammatical inference is a classical problem in machine learning [3, 12, 37]. GA and GP have been successfully applied towards the inference of formal languages [6, 10, 26, 27, 40, 44]. The use of GA and GP for stochastic language inference has also been investigated [23, 38]. A stochastic language is a formal language with a probability distribution over its set of members. For example, a stochastic regular language is a regular language whose membership conforms to some probability distribution. From one point of view, this seems like an additional burden for inference algorithms, for both the language membership and the probability distribution of its members must be ascertained. On the other hand, the use of probability distributions can be advantageous for inference algorithms for a number of reasons. First, the use of probabilities precludes the need for negative examples, since the probabilities of positive members automatically accounts for negative membership. Second, probability distributions lend an additional degree of freedom for inference, since it permits a natural means for specifying the statistical accuracy of acceptable solutions. Probabilistic language inference is naturally suited to GP, as GP is reknown for being most effective for problems that require an accurately good solution, rather than a precisely correct one with no latitude for error.

The target language used here is a probabilistic regular language called (guarded) Stochastic Regular Expressions (gSRE) [35]. It is similar to a stochastic regular language in [13], and is much like a conventional regular expression [19], except that operators have numeric fields indicating the probability that they produce results. This language is one of a number of notations that can express stochastic regular languages, the other two being stochastic finite automata (also called *Hidden Markov Models* [34]) and stochastic regular grammars [7]. These notations are theoretically equivalent in the Chomsky hierarchy, and they can be translated to each other algorithmically. There are stochastic regular languages, however, which are more clearly and concisely expressed with one notation over another. A simple gSRE expression may be equivalent to a quite complex stochastic regular grammar. For example, considering conventional regular languages, the regular expression $01^* + 1$ translates to a 10–state finite automaton [19]. Although the finite automaton recognizes the identical regular language, its complexity makes it much less transparent than the concise regular expression. Therefore, linguistic clarity of a denotation impacts the effective inference of a solution, since more complex denotations naturally require more effort by an inference algorithm.

gSRE has the following syntax. Let α range over alphabet Σ^*, E range over gSRE expressions, n range over positive integers ($0 \leq n \leq 1000$), and f range over decimal values with a precision of 2 decimal places ($0 \leq f < 1.00$). The syntax of gSRE is recursively defined as:

$$E \ ::= \ \alpha \ \mid \ \sum_i E'_i(n_i) \ \mid \ E_1 : E_2 \ \mid \ E^{*f}$$

With no loss of generality, the empty string ϵ is not included in the alphabet. The language for an expression E is $L(E)$. The meaning of the above operators is as follows.

1. Atomic action α: The action α is generated with a probability of 1.
2. *Guarded Choice* $\sum_i E'_i(n_i)$, where $E' = (\alpha_i : E_i)$ or $E' = \alpha_i$, and $\forall i \neq j : \alpha_i \neq \alpha_j$: Each term in the choice expression is prefixed with a unique atomic action that is found nowhere else in the expression. This makes guarded choice deterministic, and it contributes to smaller expressions during GP processing as compared to nondeterministic choice. Each term E'_i has a probability

   $$\frac{n_i}{\sum_j n_j}.$$

   For example, given the expression $E_1(3) + E_2(5)$, the term E_1 has a probability of 3/8 and E_2 has a probability of 5/8. The probability of the entire expression is the product of the probability of the chosen term with the probability of the expression encapsulated within it.
3. *Concatenation* "$E_1 : E_2$" : Term E_1 is interpreted, followed by that of E_2. The overall probability is the product of the probabilities for expressions E_1 and E_2.
4. *Kleene Closure* E^{*f} : Term E can be repeatedly executed 0 or more times. Each iteration occurs with a probability of f multiplied by the probability of expresion E. The probability of E not iterating is $1 - f$.

A given gSRE expression denotes a set of strings belonging to its language, as well as a probability distribution for that language. When interpreting a gSRE expression, a probability field is also maintained, as is informally explained in the above operator descriptions. For example, consider the expression

$$E \ = \ (a : b^{*0.6})(2) + c^{*0.1}(3)$$

The string c is a member of $L(E)$, and has a probability of 0.054 (the term with c can be chosen with a probability of 0.6; then that term iterates once with a probability of 0.1; finally the iteration terminates with a probability of $1 - 0.1 = 0.9$, giving an overall probability of $0.6 \times 0.1 \times 0.9 = 0.054$). The string abb is also a member of $L(E)$, and has a probability of $0.4 \times 0.6 \times 0.6 \times 0.4 = 0.0576$. The string ba is not a member of $L(E)$, and its probability is 0.

4.2 DCTG Definition of gSRE Syntax

```
expr ::= guardedexpr.                                          %  1
expr ::= choice^^A, { A^^construct(CL), unique_guards(CL) }.   %  2
expr ::= expr, expr.                                           %  3
expr ::= expr^^A, { not A^^iter }, probval                     %  4
     <:>
     iter.
choice ::= guardedexpr, intval, guardedexpr, intval.           %  5
choice ::= guardedexpr, intval, choice.                        %  6
guardedexpr ::= [A], {action_val(A)}.                          %  7
guardedexpr ::= [A], {action_val(A)},  expr.                   %  8
intval ::= [N], { is_an_integer(N) }.                          %  9
probval ::= [R], { is_a_probability(R) }.                      % 10
```

Fig. 4 DCTG Syntactic Definition of gSRE

Fig. 4 is the DCTG encoding of the gSRE grammar. To elucidate the grammar, the DCTG is simplified by the removal of all the semantic rules (other than the last rule for `expr`), as well as most of the variables used by these semantic rules.

The rules function as follows. In rules 1 through 4, the `expr` rules define top–level gSRE expressions, which may be guarded expressions (in order to include actions), choice, concatenation, or Kleene closure. In rule 2, a syntactic constraint is used to enforce guarded choice (unique prefixes in choice terms). The reference to the semantic goal `construct` of nonterminal `choice` is one which constructs a list of all the prefixes of terms in the choice expression. Then the call to the Prolog goal `unique_guards` takes this list, and succeeds if all these prefixes are unique. Otherwise, it will fail, and the parse tree will fail due to an instance of duplicate prefixes.

Rule 4 encodes an additional syntactic constraint: directly nested iterations are prohibited. For example, $(a^{*p})^{*q}$ is disallowed. This is similar to the common practice with conventional regular expressions to convert $(a^*)^*$ to a^*. Preliminary experiments found that nested iterations were commonly constructed during GP, resulting in program bloating. This dramatically slowed down the interpretation of expressions, while at the same time generating minute probabilities due to the compounded application of the iterative probability. For example, consider the expression,

$$((a^{*0.5})^{*0.5})^{*0.5}.$$

This expression generates the string *aaa* using 5 different combinations of the nested iteration operators, and with an overall probability of only 0.008. Further nesting creates combinatorially more iterations to compute, with even less significant probabilities. These expressions become a primary vehicle for intron bloat. Literally any expression can be used within the nested iteration with no adverse effect on the language generated, because the overall probability of the iteration is inconsequential anyway. Disallowing such expressions significantly speeds processing, with no negative effect on evolution. Rule 4 uses a simple constraint on the semantics of the expression embedded in its iteration: if the embedded expression is also an iteration – and only rule 4 is labelled by this `iter` designation – then the rule forces failure.

Rules 5 and 6 define choice, while 7 and 8 define guarded expressions. The term `[A]` is the DCTG notation for a terminal, and this succeeds in rules 7 and 8 if that terminal is a member of the alphabet. Finally, rules 9 and 10 define integers and floating point values, to be used in choice probabilities and iteration respectively.

The DCTG's ability to parameterize context–sensitive grammatical constraints permits a concise implementation. To see this, it is worth examining the contributions of the semantics in rules 2 and 4 in more detail. In rule 4, the semantic check for nested iteration is an example of the use of context–sensitive information (whether the expression nested in that rule is iterative or not) determining the legitimacy of the parse tree. Without the use of this semantic parameterization of this check, a more complex grammar is required. For example:

```
expr ::= noniter.
expr ::= iter.
noniter ::= guardedexpr.
noniter ::= choice, ....
noniter ::= expr, expr.
iter ::= noniter, probval.
```

This is clearly more involved than the simple context–sensitive check done in Fig. 4.

With rule 2, the alternative is even more unwieldy. Separate sets of rules are required in order to denote which actions have or have not been processed in the guarded expression. For example, if $\Sigma = \{a, b, c\}$, then the following non–DCTG attribute grammar is equivalent to the processing of guarded choice done in Fig. 4:

```
choice ::= guard_a, intval, guard_not_a, intval.
choice ::= guard_a, intval, choice_not_a.
choice ::= guard_b, intval, guard_not_b, intval.
choice ::= guard_b, intval, choice_not_b.
choice ::= guard_c, intval, guard_not_c, intval.
choice ::= guard_c, intval, choice_not_c.
choice_not_a ::= guard_b, guard_c.
choice_not_a ::= guard_c, guard_b.
choice_not_b ::= guard_a, guard_c.
choice_not_b ::= guard_c, guard_a.
choice_not_c ::= guard_a, guard_b.
choice_not_c ::= guard_b, guard_a.
guard_not_a ::= guard_b.
guard_not_a ::= guard_c.
guard_not_b ::= guard_a.
guard_not_b ::= guard_c.
guard_not_c ::= guard_a.
guard_not_c ::= guard_b.
```

Obviously, this strategy is untenable for grammars with even small alphabets.

4.3 DCTG Definition of gSRE Operational Semantics

```
choice ::= guardedexpr^^A1, intval^^B1, guardedexpr^^A2, intval^^B2
<:>
(recognize(S, S2, Sum, PrSoFar, Pr) ::-
        B1^^construct(Val),
        Pr2 is (Val/Sum)*PrSoFar,
        A1^^recognize(S, S2, Pr2, Pr)),
(recognize(S, S2, Sum, PrSoFar, Pr) ::-
        B2^^construct(Val),
        Pr2 is (Val/Sum)*PrSoFar,
        A2^^recognize(S, S2, Pr2, Pr)).
```

Fig. 5 Operational Semantics of Choice Operator (excerpt)

From section 4.1, it should be evident that gSRE has a fairly straight–forward operational semantics. Since its operational semantics are compositional, each operator's operational semantics is conveniently encoded within a semantic field of the DCTG. During fitness evaluation, the GP will test whether gSRE expressions can recognize different example strings of the target language. Hence the operational semantics are encoded so that expression interpretation attempts to recognize the membership of strings, and produce their corresponding probabilities if so recognized. Given a string to recognize, the actual implementation finds the probability of the longest prefix recognized by an expression.

A complete account of the implementation of gSRE's operational semantics is beyond the scope of this paper. To give a flavour of the implementation, Fig. 5 contains the semantics for rule 5 of Fig. 4. This rule is included in any parse tree containing choice, as it terminates the tail recursion used in rule 6. The semantics of this rule applies the probability of choosing one of the two choice terms within the computation of the overall probability. The rule `recognize(S, S2, Sum, PrSoFar, Pr)` has 5 arguments: the string `S` at the start of processing of this choice expression; the string `S2` after processing (`S2` is either equal to `S` or a suffix of it); the `Sum` of the probability values terms in the choice list (ie. the overall denominator value); the probability `PrSoFar` computed so far while processing the current string (other expression components may have read earlier prefixes of the example string, yielding this probability); and the final computed probability `Pr` after processing `S` is completed.

There are two rules for `recognize`, and each rule processes one of the terms from the pair of choice expressions. The call to `construct` retrieves the actual integer value from the probability field for that term. The probability for that term is calculated, and multiplied by the overall probability so far. This new intermediate probability is then passed to the recognition semantics for the expression embedded in that choice term. During processing, both rules will be invoked – the first followed by the second – because there are multiple ways a string can be recognized by an gSRE expression. The operational semantics for gSRE will exhaustively try all rules until the string is completely recognized, and no alternative derivations of the gSRE expression are possible. All the probabilities obtained for these difference derivation paths are collected and summed elsewhere, to yield an overall probability for that string (or its longest prefix).

An example execution is as follows. Consider the expression $E(2) + F(3)$, where E and F are gSRE expressions. Elsewhere it is determined that the denominator for this expression is 5. Assume that earlier interpretation of other terms has computed a probability so far of 0.5. Then `recognize(S, S2, 5, 0.5, Pr)` will first process the $E(2)$ choice term. The probability is updated to be $2/5 \times 0.5 = 0.2$, and then the term E is interpreted, using the call $A1$^^`recognize(S, S2, 0.2, Pr)`. Its successful interpretation will return the overall probability `Pr` for this term. The interpretation of the $F(3)$ term is similar, except that an intermediate probability 0.3 is computed.

4.4 Other Experiment Details

Table 1 Experimental Parameters

Parameter	**Value**
Target language	gSRE
Terminals	a, b, c
DCTG root	expr
Fitness function	modified χ^2 on random test set
Generation type	steady-state
Initial population size	750
Running population size	500
Unique population members	yes
Maximum generations	50
Probability of crossover	0.90
Probability of mutation	0.10
Probability internal crossover	0.90
Probability terminal mutation	0.75
Probability numeric mutation	0.50
Numeric mutation range	± 0.1
Max reproduction attempts	3
Initial population shape	ramped half&half
Min/max depth initial popn.	6, 12
Max depth offspring	24
Tournament size, selection	2
Tournament size, replacement	3
Test set size	1000
Approx. max. test string size	20
gSRE probability limit	0.00001

The task is to evolve a gSRE expression that corresponds to the language for the following target expression:

$$a : c^{*0.5} : a\ (1) \quad + \quad b : c^{*0.5} : b\ (1)$$

The fitness strategy used is to test how well an expression can recognize a random test set of 1000 strings generated for this target description. The test set is regenerated randomly before each new generation. The entries are counted, and the resulting test set is a list of string values and their frequencies.

Each test set string t_i is given to the gSRE interpreter as encoded in the DCTG, and its membership probability is computed. A modified χ^2 formula [33] is used in the fitness evaluation formula:

$$Fitness = \sum_{t_i \in T} \begin{cases} \dfrac{(d_i - (Pr(t_i) * N))^2}{d_i} & : Pr(t_i) \geq 0 \\ \left(1 + \dfrac{|t_i| - |maxpref_i|}{|t_i|}\right) \cdot d_i & : Pr(t_i) = 0 \end{cases}$$

where d_i is the frequency of example t_i in test set T, $N = |T|$, and $maxpref_i$ is the maximum prefix of t_i recognized. The first term is the χ^2 formula, and it is used when the example string t_i is completely recognized. The second formula is used when only a prefix of t_i is recognized. Its value is inversely proportional to the size of the prefix recognized. Should none of t_i be recognized, then this value becomes $2 \cdot d_i$ (a normal χ^2 formula would use just d_i). This prefix scoring gives credit to expressions that recognize portions of the examples, which helps drive evolution towards expressions that recognize complete examples.

Other GP parameters are listed in Table 1, and most are self–explanatory from the GP literature. Mutation is specialized so that it can mutate the numeric terminals found throughout gSRE expressions. The probability of numeric mutation states that 50% of the time numeric fields will be selected for mutation, and the mutation range specifies that a numeric value will be randomly perturbed ±10% of its value. Failed reproduction operations (offspring too large, non-terminals not matching in selected substrings,...) will be reattempted 3 times, after which new parents will be used. The gSRE probability limit specifies a lower cutoff probability during gSRE recognition. When the probability becomes lower than this value, recognition using the current expression derivation path is halted, and a new derivation is attempted. This boosts efficiency by pre-empting inconsequential derivations.

4.5 Results

Table 2 summarizes the experiment results. The values in the table are computed from the 25 runs for a fixed test set. Note that there is a significant degree of variation in the fitness of individuals for different test sets, because the randomly generated test set varies between generations and runs. The average test set χ^2 gives an indication of how much different test sets vary with each

Table 2 Summary

Total runs	25	
# unique examples	19	
Avg. test set χ^2 (50 cases)	30.68	
Fitness	min	12.67 (χ^2=12.67)
	max	328.6 (χ^2=314.23)
	avg	150.54 (χ^2=148.26)

other. Therefore, although the overall average solution obtained for the set of runs is not outstanding, the best solutions are very good. Fig. 6 shows the average performance of the 25 runs.

The best solution, with a fitness of 12.67, has the following genotype:

```
node(expr,[node(choice,[node(guardedexpr,[[a],node(expr,[node(expr,
[node(expr,[node(guardedexpr,[[c]],6)],0), node(probval,[[0.49]],9)],3),
node(expr,[node(choice,[node(guardedexpr,[[a]],6),node(intval,[[819]],8),
node(guardedexpr,[[b],node(expr,[node(expr,[node(choice,
[node(guardedexpr,[[c]],6),node(intval,[[34]],8),node(guardedexpr,[[b]],6),
node(intval,[[890]],8)],4)],1),node(expr,[node(choice,
[node(guardedexpr,[[b]],6),node(intval,[[435]],8),node(guardedexpr,[[c]],6),
node(intval,[[341]],8)],4)],1)],2)],7),node(intval,[[6]],8)],4)],1)],2)],7),
node(intval,[[532]],8),node(guardedexpr,[[b],node(expr,[node(expr,
[node(expr,[node(guardedexpr,[[c]],6)],0),node(probval,[[0.49]],9)],3),
node(expr,[node(choice,[node(guardedexpr,[[a]],6),node(intval,[[6]],8),
node(guardedexpr,[[b]],6),node(intval,[[990]],8)],4)],1)],2)],7),
node(intval,[[553]],8)],4)],1)
```

Translated into gSRE, this is:

$$\begin{aligned} & a : c^{*0.49} : (a(819) + (b : (c(34) + b(890)) : (b(435) + c(341)))(6))(532) \\ & \;+\; b : c^{*0.49} : (a(6) + b(990))(553) \end{aligned}$$

A further translation into a more readable form, where the probabilities over choice are directly given, is:

$$\begin{aligned} & \mathbf{.49}\ (a : c^{*.49} : (\mathbf{.99}\ a + \mathbf{.01}\ (b : (\mathbf{.03}\ c + \mathbf{.97}\ b) : (\mathbf{.56}\ b + \mathbf{.44}\ c)))) \\ & \;+\; \mathbf{.51}\ (b : c^{*0.49} : (\mathbf{.001}\ a + \mathbf{.99}\ b)) \end{aligned}$$

This is further simplified by removing terms with statistically insignificant probabilities:

$$\mathbf{.49}\ (a : c^{*.49} : a) \quad + \quad \mathbf{.51}\ (b : c^{*0.49} : b)$$

Fig. 6 Fitness curves (avg 25 runs)

The above is essentially the target language expression. Note that the removed expression terms are *intron* material – genetic information that does has no ill effect on the fitness of an individual, other than bloating its size.

A poor solution with a fitness of 263 is:

$$(b : c^{*.5}(368) + (a : (c) * 0.48)(454)) : (a(827) + b(760))$$

Transforming it in the manner above, it becomes:

$$(\mathbf{.45}\ b : c^{*.5}\ +\ \mathbf{.55}\ a : c^{*.48}) : (\mathbf{.52}\ a\ +\ \mathbf{.48}\ b)$$

The left–side of the top–level concatentation term accurately processes the prefixes of the target language. However, the right–side is only partially correct. For example, the string *bca* is inferrable with a significantly high probability of 0.0585. This expression thus differs from normal intron material, which generates erroneous strings with very low probabilities.

§5 OTHER WORK

DCTG-GP is inspired by the LOGENPRO system[43]. LOGENPRO is also implemented in a logic programming language, and uses the DCG logic grammar paradigm[32]. LOGENPRO's crossover procedure blindly selects nodes for potential exchange, and only succeeds if it is determined that the offspring are grammatically correct. Even though crossover is usually inexpensive for smaller–sized trees, this is somewhat wasteful, as a significant proportion of crossover operations will fail. This is overcome in DCTG-GP by only selecting similarly labelled node types for crossover. Although the details of the system representation for programs are not given in[43], it appears that a logic–program representation is used, since the logic grammar argument structure must be maintained during all crossover and reproduction operations. DCTG-GP uses a simplified tree structure that denotes the essence of the grammar tree only. If semantic components are required for verification after crossover or mutation, the DCTG clauses for the grammar is then interpreted in unison with the grammar encoding for the program. This permits semantic execution to be circumvented when it is not required.

One major advantage of a DCTG over LOGENPRO's DCG is the additional semantic expressiveness inherent in a DCTG. The motivation behind DCTG's in the first place is that a DCG cannot encode substantial semantic information without the grammar becoming overly complex and unwieldy. A DCTG overcomes the representational weaknesses of DCG's by encoding the semantics into modularly separate areas, while still tying them together with the rules to which they pertain. Argument notations permit semantic information from various goals in the syntactic rule to be extracted and used as required. The net result is that more complex semantics, including entire language interpreters, are encodable with a DCTG. DCTG semantic definitions are also practical for incorporating syntactic constraints into the grammar of the target language. Although such syntactic constraints might be definable via syntactic rules, the use of semantics for this purpose can greatly simplify the grammar, as is seen in Section 4.2.

Gruau uses CFG's to enforce syntactic properties in genetic programs[16]. Gruau suggests that, if a user knows a syntactic constraint that a solution must have, she or he should encode this property in the grammar of the target language's search space. Hence the semantics of the search space is constrained by the user in so far as the semantic constraints are effectively controlled by syntactic constraints. This is not as general a means for controlling search as is potentially available with logic grammars such as DCTG's, which allow more generalized context–sensitive constraints.

Gruau's system does recursive call counting to ensure that program trees do not exceed given depth limits. It is unclear, however, how he ensures that rules selected during program generation do not result in unduly small trees, since he does not appear to identify the termination properties of rules (ie. whether a rule always terminates, or can generate a subtree less than a given depth). Hence it is likely that programs in the initial population tend to be shallow in depth, given the lack of control of generated program sizes. As in DCTG-GP, crossover in his system only selects subtrees having the same nonterminal identifier, which guarantees grammatical correctness. Additionally, Gruau transforms grammars into normal forms, which increases the likelihood of permissible crossover possibilities than is possible for non–transformed productions. Gruau uses a binary encoding for the grammars, since grammatical representations can be resource expensive. It could be argued that such an encoding may hinder execution speed at the expense of cheap memory.

Whigham uses CFG's in GP[41, 42)]. As with Gruau's system and DCTG-GP, only nodes having the same nonterminal identifier are selectable for crossover exchange. Whigham controls the size of generated trees by requiring that the user manually encode each production with a number indicating the minimal tree depth possible with that rule. DCTG-GP automatically determines this information. Since Whigham does not analyze the termination characteristics of rules, the initial population will favour smaller trees.

Some researchers suggest using a linear representation of programs, in which genes are numbers that map to grammar rules[11, 36)]. Gene mapping is performed in a way so that only legal grammatical trees are generated; translations to erroneous grammatical structures are ignored. This denotation is arguably more akin to a GA with variable–length chromosome than the conventional tree–based GP denotation, as a translation step from a basic genotype into the grammar–tree phenotype must be undertaken. With this approach, grammatical correctness does not need to be preserved during reproduction, since erroneous chromosome mappings are circumvented during gene translation. In addition, since the grammar rules are not used for tree generation (chromosomes are lists of numbers), the depths of trees and termination characteristics of grammar rules do not need to be considered.

Other work using grammars in GP is as follows. Geyer–Schulz has implemented various GA and GP systems that use grammars[14, 15]. He concentrates on studying issues with the syntactic context-free grammar representations for use within GP systems, and assumes that attribute grammars, compiler-compilers and other tools are available to implement the semantics of the languages. Jacob uses grammatical GP to encode L-systems for use in modelling natural phenomena[22]. L-systems are context-free grammars whose terminals have semantic interpretations usually associated with graphical meanings. The above works[14, 15, 22] are related to DCTG-GP in that their use of context-free grammars are equivalent to the syntactic portion of DCTG rules.

Hussain and Browse use attribute grammars and GP to evolve neural networks[20]. Their use of attribute grammars is closely related to DCTG-GP, since the semantic component of DCTG's is a logical implementation of an attribute grammar. The main difference between their attributes and DCTG semantics is one of implementation style. DCTG semantics are tied directly to individual grammar rules, permitting direct access to portions of the parse tree. Traditional attribute grammars as in [20] define attributes separately from the grammar rules, which they are referenced within the grammar at appropriate locations. Their attribute grammar uses a conventional Pascal–like language for defining semantics, while DCTG-GP uses logic programming.

Work related to grammatical GP is that of GP with data typing[17, 24, 31]. Data typing constrains solutions with respect to the range of argument values passed to program constructs. Therefore, it is a subset of the scope of syntactic constraint done within more general grammatical GP systems.

The problem of generating random trees having particular size and shape constraints has been investigated elsewhere[2]. Iba gives a linear algorithm for generating random grow trees for GP[21]. He shows that the basic grow tree generation algorithm of[24] does not produce a uniform distribution of trees across a range of depths. His tree generation algorithm generates trees with a more uniform distribution of depths, and this is shown to have a positive influence on evolution. The algorithm is designed around Koza's basic CFG with one nonterminal, and it is not directly applicable to more complex CFG's.

Böhm and Geyer-Schulz address random tree generation for grammatical GP[5]. Rather than consider the exact shape or depth of trees, their metric for tree size is the number of derivation steps required by the grammar to derive a tree. A combinatorial view of context–free languages is taken, in which trees that can be generated in a set number of derivation steps are said to belong to the same partition of the language's search space. Their algorithm uses this knowledge about word partitions to randomly derive a tree within a particular partition. By deriving trees within different partitions, a variety of trees can be randomly generated. The implementation of their algorithm is nontrivial, and the use of partition spaces does not guarantee depth or breadth characteristics of trees. In comparison, the algorithms given in this paper are simple to implement, and result in acceptably variable random tree shapes. Grammar analyses is related to work in abstract interpretation, which is applied to programs in order to automatically determine such things as termination and typing characteristics[9].

§6 CONCLUSION

This paper has discussed the implementation of a logic–based GP system that uses a DCTG to encode the syntax and semantics of the target language. DCTG-GP builds upon earlier approaches, extending LOGENPRO system[43] by using DCTG's, and using established techniques for maintaining grammar correctness during reproduction[16, 41]. The system analyzes a user's grammar for information about the termination and minimal–depth characteristics of rules, which is required for effective random tree generation. Recently, DCTG-GP has been used successfully elsewhere in research investigating the use of fitness sharing in genetic programming[29, 30]. In that work, grammars are defined in DCTG-GP that permit the evolution of list membership and multiplexer programs.

As illustrated in our example experiment, two practical uses of a DCTG's semantics are to simplify the grammatical definition of the language, and to encode the operational semantics of the target language. Many languages have compositional operational semantics, and defining an interpretation procedure for them within a DCTG is straight–forward. Although syntactic problem constraints can be defined with the grammatical definition of the language, this can be unwieldy and complex, as will be the resulting parse trees for the programs. The DCTG can parameterize syntactic properties of the grammar with semantic rules, which substantially simplifies the grammar. Of course, the DCTG semantics can encode much more complex semantic constraints of programs, for example, actually interpreting partial subtrees of the programs in order to determine their acceptability. This must be done with caution, however, as the benefits in pruning the search space in this manner may be outweighed by the resulting overhead in performing reproduction. In addition, if the search space is overconstrained, evolution will not be successful. Further investigation on the use of semantic constraints during GP is necessary.

There are a number of enhancements possible. Gruau's normal form for productions[16] is worth consideration. For example, one normal form transformation ensures that all terminals have nonterminal ancestors. After this and other correctness–preserving transformations are performed on the grammar, there are more robust opportunities for sensible node selection for directed crossover. Whigham's experiments[41, 42] in evolving new productions during evolution are intriguing, as they suggest that grammars themselves can be evolved in order to discover more useful instances. Further work is necessary in this topic, and to this end, attention should be directed towards the substantial body of work in grammar evolution[26, 28, 40].

Both Prolog and Lisp are AI languages, whose symbolic paradigms make them ideal for genetic programming implementations. However, being interpreted, they are very slow compared to GP systems programmed in compiled languages. It is interesting to consider a DCTG–style interface for a GP system implemented in a lower–level but faster language than Prolog, such as C++.

Acknowledgement: Helpful comments from anonymous referees are gratefully acknowledged. This work is supported though NSERC Operating Grant 138467-1998.

References

1) H. Abramson and V. Dahl. *Logic grammars.* Springer-Verlag, 1989.
2) L. Alonso and R. Schott. *Random Generation of Trees.* Kluwer Academic Publishers, 1995.
3) D. Angluin. Computational Learning Theory: Survey and Selected Bibliography. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 351–369. ACM Press, 1992.
4) W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, 1998.
5) W. Bohm and A. Geyer-Schulz. Exact Uniform Initialization for Genetic Programming. In R.K. Belew and M. Vose, editors, *Foundations of Genetic Algorithms 4*. Morgan Kaufmann, 1997.
6) S. Brave. Evolving deterministic finite automata using cellular encoding. In John R. Koza *et al*, editor, *Proc. Genetic Programming 1997*, pages 39–44, Stanford University, CA, USA, 1997. Morgan Kaufmann.
7) E. Charniak. *Statistical Language Learning.* MIT Press, 1993.

8) W.F. Clocksin and C.S. Mellish. *Programming in Prolog (4th ed).* Springer-Verlag, 1994.

9) P. Cousot. Abstract Interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

10) P. Dupont. Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG method. In *2nd Intl. Coll. on Grammatical Inference and Applications*, pages 236–245. Springer-Verlag, 1994.

11) J.J. Freeman. A Linear Representation for GP using Context Free Grammars. In J.R. Koza *et al.*, editor, *Proc. Genetic Programming 1998*, pages 72–77. Morgan Kaufmann, 1998.

12) K.S. Fu and T.L. Booth. Grammatical Inference: Introduction and Survey – Part I. *IEEE Transactions on Systems, Man, and Cybernetics*, 5(1):95–111, January 1975.

13) V.K. Garg, R. Kumar, and S.I Marcus. Probabilistic Language Framework for Stochastic Discrete Event Systems. Technical Report 96-18, Institute for Systems Research, University of Maryland, April 1996. http://www.isr.umc.edu/.

14) A. Geyer-Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning.* Studies in Fuzziness and Soft Computing, v.3. Physica-Verlag, Heidelberg, 1996. 2nd revised edition.

15) A. Geyer-Shulz. The Next 700 Programming Languages for Genetic Programming. In John R. Koza *et al*, editor, *Proc. Genetic Programming 1997*, pages 128–136, Stanford University, CA, USA, 1997. Morgan Kaufmann.

16) F. Gruau. On Using Syntactic Contraints with Genetic Programming. In P.J. Angeline and K.E. Kinnear, editors, *Advances in Genetic Programming II*, pages 377–394. MIT Press, 1996.

17) T.D. Haynes, D.A. Schoenefeld, and R.L. Wainwright. Type Inheritance in Strongly Typed Genetic Programming. In P.J. Angeline and K.E. Kinnear, editors, *Advances in Genetic Programming II*, pages 359–375. MIT Press, 1996.

18) J.H. Holland. *Adaption in Natural and Artificial Systems.* MIT Press, 1992.

19) J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

20) T.S. Hussain and R.A. Browse. Attribute Grammars for Genetic Representations of Neural Networks and Syntactic Constraints of Genetic Programming. In *AIVIGI'98: Workshop on Evolutionary Computation*, 1998.

21) H. Iba. Random Tree Generation for Genetic Programming. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature IV, Proc. of the International Conference on Evolutionary Computation*, pages 144–153. Springer-Verlag, 1996.

22) C. Jacob. Evolving Evolution Programs: Genetic Programming and L-Systems. In J.R. Koza *et al*, editor, *Proc. Genetic Programming 1996*, pages 107–115. MIT Press, 1996.

23) T.E. Kammeyer and R.K. Belew. Stochastic Context-free Grammar Induction with a Genetic Algorithm Using Local Search. In R.K. Belew and M. Vode, editors, *Foundations of Genetic Algorithms IV.* Morgan-Kaufmann, 1997.

24) J.R. Koza. *Genetic Programming.* MIT Press, 1992.

25) J.R. Koza. *Genetic Programming II.* MIT Press, 1994.

26) M.M. Lankhorst. Grammatical Inference with a Genetic Algorithm. *Proceedings of the 1994 EUROSIM Conference on Massively parallel Processing Applications and Development*, pages 423–430, 1994.

27) T. Longshaw. Evolutionary learning of large grammars. In J.R. Koza *et al*, editor, *Proc. Genetic Programming 1997*, pages 406–409, Stanford University, CA, USA, 1997. Morgan Kaufmann.

28) S. Lucas. Structuring chromosomes for context-free grammar evolution. In *Proceedings 1st International Conference on Evolutionary Computation*, pages 130–135. IEEE Press, 1994.

29) R.I. McKay. Fitness Sharing in Genetic Programming. In D. Whitley *et al.*, editor, *Proc. GECCO 2000.* Morgan Kaufmann, 2000.

30) R.I. McKay. Partial Functions in Fitness-Shared Genetic Programming. In A. Zalzala, editor, *Proc. Congress on Evolutionary Computation.* IEEE Press, 2000.

31) D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

32) F.C.N. Pereira and D.H.D Warren. Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13:231–278, 1980.

33) W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 2 edition, 1992.

34) L.R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.

35) B.J. Ross. Probabilistic Pattern Matching and the Evolution of Stochastic Regular Expressions. *Applied Intelligence*, 2000. In press.

36) C. Ryan, J.J. Collins, and M. O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In W. Banzhaf *et al.*, editor, *Proc. First European Workshop in Genetic Programming (EuroGP-98)*, pages 83–96. Springer-Verlag, 1998.

37) Y. Sakakibara. Recent Advances of Grammatical Inference. *Theoretical Computer Science*, 185:15–45, 1997.

38) M. Schwehm and A. Ost. Inference of Stochastic Regular Grammars by Massively Parallel Genetic Algorithms. In *Proc. 6th Intl. Conf. on Genetic Algorithms.* Morgan-Kaufmann, 1995.

39) SICS. *SICStus Prolog V.3 User's Manual*, June 1995. http://www.sics.se/isl/sicstus.html.

40) B. Svingen. Learning Regular Languages Using Genetic Programming. In J.R. Koza *et al*, editor, *Proc. Genetic Programming 1998*, pages 374–376. Morgan Kaufmann, 1998.

41) P.A. Whigham. Grammatically-based Genetic Programming. In J.P. Rosca, editor, *Proceedings Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 31–41, 1995.

42) P.A. Whigham. Inductive Bias and Genetic Programming. In A.M.S. Zalzala, editor, *1st International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA)*, pages 461–466, 1995.

43) M.L. Wong and K.S. Leung. Evolutionary Program Induction Directed by Logic Grammars. *Evolutionary Computation*, 5(2):143–180, 1997.

44) H. Zhou and J.J. Grefenstette. Induction of Finite Automata by Genetic Algorithms. In *Proc. 1986 IEEE Intl. Conference on Systems, Man, and Cybernetics*, pages 170–174, Atlanta, GA, 1986. IEEE Press.